Survey on Buffer Overflow Attacks and Countermeasures

Tim Werthmann
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
tim.werthmann@ruhr-uni-bochum.de

June 26, 2006

Abstract

In this paper we point out how buffer overflow attacks influence data processing. Further we discuss recent countermeasures presented in the scientific literature and that it takes considerable effort to prevent buffer overflows. On the one hand static methods produce false positives and negatives and on the other hand instrumentation methods have a lot of overhead. However, with a little more research and some optimizations of the architecture instrumentation methods prevent most of the buffer overflow attacks. Stack based methods do not protect against all attacks and some protections can be bypassed. Hardware methods provide protection with little overhead, but architectural changes have to go deep to prevent all known buffer overflow attacks. Making the stack non-executable does not solve the buffer overflow problem either, and blocking buffer overflows based on code injections does not prevent most of the attacks and the assumptions must be met, else valid transactions could be blocked.
It is not very surprisingly that all buffer overflows are caused by errors made by programmers and/or data processing. This leads to the conclusion that data processing has to be renewed consequently. For this purpose the C standard should be updated, because the instrumentation methods show that enhanced C applications successfully thwart buffer overflow attacks.

1 Introduction

In 1988, when the Morris Worm caused the first large scaled attack based on buffer overflows, one of the most wide-spread attacks of the cyberworld awake. 18 years later, buffer overflows are still on the daily schedule, and for now the attack is still not stopped nor under control. The Code Red Worm from 2001 for example caused 2.5 billion USD damage [17] and this is only one example of a large scaled buffer overflow misusing. Since C does not enforce bounds checking on certain standard functions (see Figure 1 for a partial list), it is possible to write more data to a buffer than the buffer could store. This leads to an overrun of the buffer (the so called buffer overflow), which may be used

by an attacker to modify the program state (e.g. in the execution of arbitrary code or in a program crash). The consequences of a buffer overflow are that an attacker could use a buffer overflow to inject code, and execute this code afterwards. Since every buffer overflow leads to an abnormal program behavior (the program was never supposed to run with the situation of a buffer overflow), the program should be terminated after an overflow occurred (if it is not possible to restore the complete program state before the overflow happened). These problems are well known and understood (see e.g. [12], [13], [14]) but not fully solved yet. The rest of the paper is organized as follows. In chapter 2, we observe the preliminaries regarding buffer overflow attacks. In chapter 3, we will discuss countermeasures recently presented in scientific literature and conclude the paper in chapter 4.

Function prototype	Potential problem
`strcpy(char *dest, const char *src)`	May overflow the **dest** buffer.
`strcat(char *dest, const char *src)`	May overflow the **dest** buffer.
`getwd(char *buf)`	May overflow the **buf** buffer.
`gets(char *s)`	May overflow the **s** buffer.
`fscanf(FILE *stream, const char *format, ...)`	May overflow its arguments.
`scanf(const char *format, ...)`	May overflow its arguments.
`realpath(char *path, char resolved_path[])`	May overflow the **path** buffer.
`sprintf(char *str, const char *format, ...)`	May overflow the **str** buffer.

Figure 1: Partial list of unsafe functions in the standard C library

2 Preliminaries

2.1 The IA-32 architecture

Today most computers are based on the Intel IA-32 architecture (sometimes referred to as x86 architecture). This architecture has a text/data/BSS/heap and stack segment (Figure 2 shows the segmentation scheme).

The Text segment is read-only and consists of opcodes, operation codes namely the program needs to be functional and that are executed in the program run. The Data/BSS (Block Started by Symbol) segment consists of global and static variables, where the initialized variables go to the Data segment and the uninitialized variables goto the BSS segment. The heap segment is used for dynamically allocated data structures, where the programmer is in charge to administrate them. Finally the stack segment is an abstract data structure based on the LIFO (Last In, First Out) principle, containing local variables. Objects are pushed onto the top of stack and the last pushed object can be popped off it. In our architecture the stack grows dynamically from higher to lower addresses and the heap dynamically from lower to higher addresses (see Figure 2).

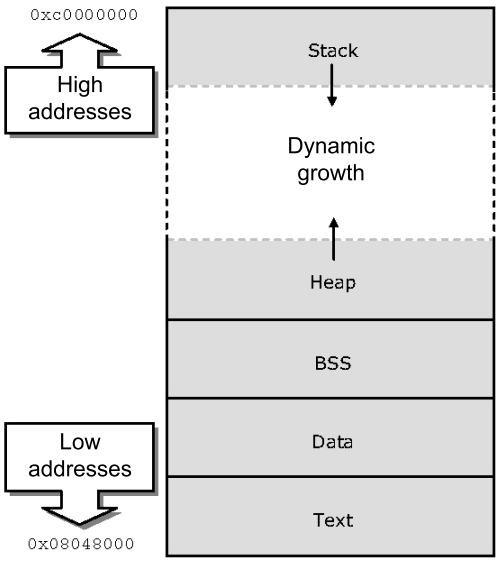

Figure 2: IA-32 segmentation sheme

For internal organization the CPU has a couple of registers, the important ones are:

- EIP (Extended Instruction Pointer)
- EBP (Extended Base Pointer)
- ESP (Extended Stack Pointer)
- General purpose Registers EAX/EBX/ECX/EDX

The EIP points to the next-to-execute op-code in the memory, it is administrated by the CPU, so only the CPU can modify its value. The ESP points to the top of stack, this way the CPU needs only two operations to work with stack data, namely push and pop. But since the value of the ESP is changed in a function, it cannot be used as reference point for indirect addressing[1]. For this purpose EBP (sometimes referred to as frame pointer) is used.
If a function is called by the assembler command "call" a new stackframe is created for that function (Figure 3). The boundaries of this frame are the EBP and the ESP. The call command first pushes the current EIP onto the stack, then the functions prologue starts. The previous EBP is pushed onto the stack, the previous ESP becomes the new EBP and then space for local variables is reserved by subtracting its size from the ESP[2] (Figure 4 shows an example function call). At the end of a function the current EBP becomes the ESP again, the EBP is restored and a "ret" (return) is executed, which restores the saved EIP from the stack to the register. This order is important, because the stack is a LIFO queue.

[1]The ESP could be used as reference point, but keeping track of the changes on the stack would result in far more overhead to the CPU

[2]If the size is not divisible by 4 (4 bytes equal 32 bit), it is round up to the next multiple of 4

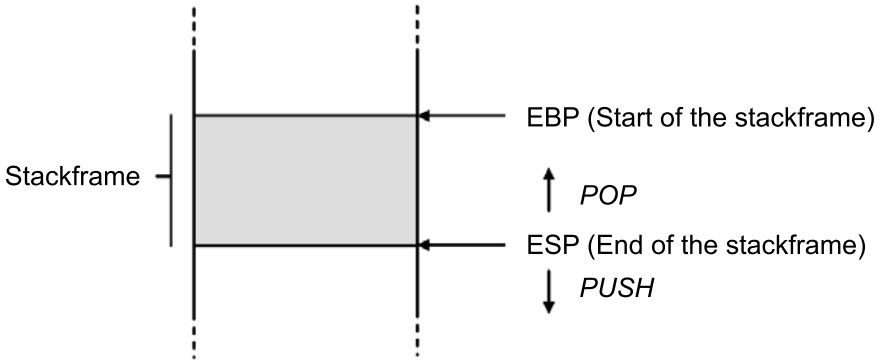

Figure 3: Example stackframe

```
int func(int b) {
  char str[8] = "string";
  int c = b;
}
int main() {
  int a = 1;
  func(a);
  return 0;
}
```

```
_func:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $12, %esp
    movl   0x80484b4, %eax
    movl   %eax, -8(%ebp)
    movl   8(%ebp), %ebx
    movl   %ebx, -12(%ebp)
    movl   %ebp, %esp
    popl   %ebp
    ret
_start:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $4, %esp
    movl   $1, -4(%ebp)
    pushl  -4(%ebp)
    call   _func
    ...
```

Figure 4: C source code and its assembler derivative

2.2 Buffer overflow variants

Today several buffer overflow attacks are known and well understood. In general every buffer that could be accessed by an attacker might be compromised if vulnerable functions are used. Such variables are located on the stack, the heap and the BSS. In most of the literature the attacks are partitioned as follows:

- Stack smashing, used to execute injected code (shellcode) or for DoS (denial of service) attacks (see e.g. [12], [13], [14], [18], [19]).
- Variable Attack, used to modify the program state (related to stack smashing).
- Heap Overflows, used to execute arbitrary code or to modify variables (see e.g. [14], [18]).

- Off-By-One, a classic programmers error, only one byte is overwritten e.g. by wrong termination of loops (see e.g. [14]).
- BSS Overflows, related to heap overflows/variable attack (see e.g. [14]).
- Signed/Unsigned Overflows, because of the twos complement, negative numbers become large if they are interpreted positive (see e.g. [14]).
- Frame Pointer Overflows, related to stack smashing, EBP is misused for indirect addressing or to damage the stackframe. (see e.g. [14])

2.3 The principles of buffer overflow attacks

We give based on stack smashing an example of the notion how buffer overflows are structured. However, we stress out that other variants mainly follow the same principle[3]. The stack smashing attack principle overwrites variables above a buffer. This causes, that all data from the beginning[4] of that buffer is corrupted. Since the EBP is overwritten, the stackframe is damaged and further execution leads to a segmentation fault (see Figure 5).

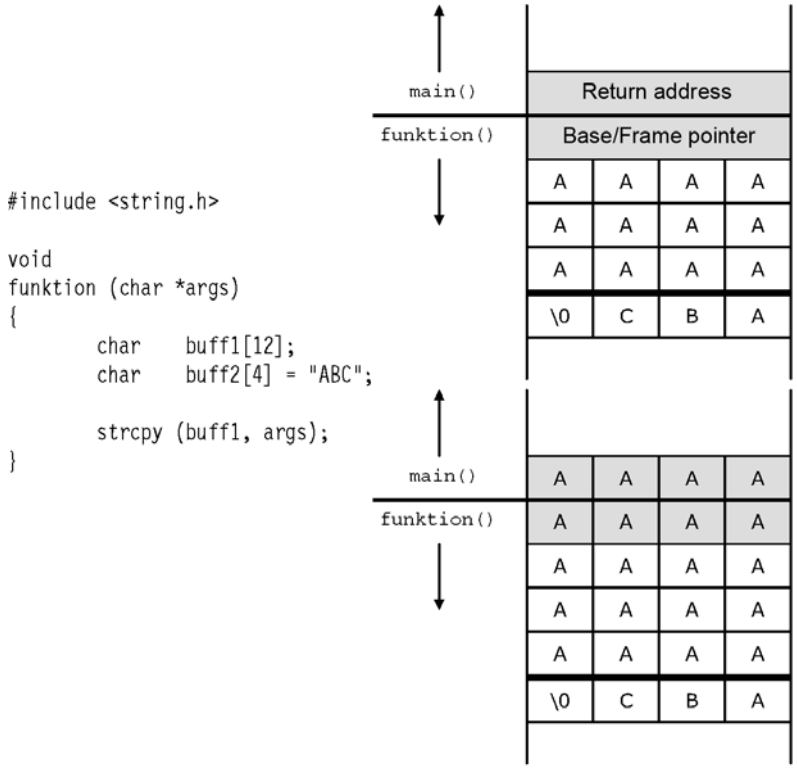

Figure 5: Use of unsafe functions and stack smashing

[3]The attacks and their severity depend on the used functions/caused errors, so this example is only schematic and easy, but could be adapted to real world attacks.

[4]If it cannot be determined what size the buffer had, the corruption starts at the beginning of that buffer, else at the end of that buffer

But the more important thing is, that the saved EIP is the only way to redirect the execution flow, since the EIP is administrated by the CPU and cannot be changed. Stack smashing can be used to point the saved EIP to injected, arbitrary code, because at a return command the EIP is restored and the code is executed.

2.4 Principles of code injection and obfuscation

If an attacker was successful in finding a buffer overflow vulnarability, he can inject a small self-made code into the program and redirect the program flow by overwriting the return address. The injection of so called shellcode[5] is a dangerous attack, because the attack might lead to get full control[6] of the affected machine. Such shellcode is bound to certain rules, it has to be small and it should not contain any termination characters (e.g. "0x00" the string terminator). The size is given by the space allocated on the stack. It is the size of the buffer plus the size of the variables/buffers above the affected buffer plus the size of the EPB and everything round up to 32 bit boundaries. The Code itself is written in assembler and after compiling the code to flat binary (meaning the assembler code is only translated one by one into hex values), all terminator characters are removed by skillfully changing the code until they all disappeared (e.g. "`mov eax, 0`" could be substituted by "`xor eax, eax`"). Figure 6 shows an example shellcode before compilation; it uses "code obfuscation" methods, which we will discuss next. Code obfuscation can be any

```
        jmp end                 # jump to the end
start:
        popl %ecx               # store eip just put on the stack
        movl %ecx, %ebx         # copy it again for later purposes
        subb $6, %cl            # point to the end of the encrypted code
xor_loop:
        movb (%ecx), %al        # read one character
        cmpb $0x90, %al         # compare to a nop
        je xor_exit             # if it is a nop we are done
        xorb $0x55, %al         # xor with 0x55
        movb %al, (%ecx)        # write back
        loop xor_loop           # loop, %ecx is decremented
xor_exit:
        nop                     # mark
        movl $16, %eax          # chown
        subl $8, %ebx           # start of string
        xorl %ecx, %ecx         # set to 0 (root)
        xorl %edx, %edx         # set to 0 (root/wheel)
        int $0x80               # call Kernel
        movl $15, %eax          # chmod
        movl $06775, %ecx       # access rights
        int $0x80               # call Kernel
        movl $1, %eax           # exit
        xorl %ebx, %ebx         # set to 0
        int $0x80               # call Kernel
        .string "sh"            # the string
end:
        call start              # put %eip on the stack
```

Figure 6: Shellcode written in assembler, using XOR code obfusciation

alternative code that makes its analysis more difficult. There are methods of

[5]Named by its target, open a (root) shell

[6]Under the assumption that the shell has administrator/root privileges

obfuscation that are polynomial but exponential in deobfuscation [2]. Some important methods are:

- Encryption, using XOR or other techniques to camouflage injected code.
- Splitting, split up arrays/variables to thwart flow analysis and confuse reader.
- Insertion of garbage data, trap disassembling of code and confuse reader.

All obfuscations (so far) can be deobfuscated, but the thing about self modifying (polymorphic) shellcodes is, that signatures are not working. If the same shellcode is obfucated using XOR (from the standard ASCII chars) it would take 255 signatures to be stored for one code. Other methods enlarge the amount of signatures to store drastically, which makes finding shellcodes by signatures a bad idea.

3 Ideas of countermeasures

In this section, we will discuss recent countermeasures discussed in the scientific literature and point out some shortcomings. Commonly countermeasures are classified into the following methods:

- Stack based. Adding redundant information/routines to protect the stack or parts of the stack.
- Instrumentation. Replacing of standard functions/objects like pointers to equip them with "tools".
- Hardware based. Architecture checks for certain illegal operations and modifications.
- Static. Checking the source code/compiled binary for known vulnerable functions, do flow analysis, check for correct boundaries, use of heuristics.
- Operation system based. Declare the stack as non-executable to prevent code execution.
- Data analysis with filtering. Analyse incoming packets for code injections and discarding if code is found.

Next we will discuss these approaches more detailed by observing methods that were introduced in the past.

3.1 Stack based

3.1.1 Stackguard

Cowan et al. propose in [3] a simple approach to protect programs against stack smashing and with little modification against EBP overflows. This is

achieved by a compiler extension that adds so called *canary* values[7] before the EIP saved at the function prologue (see Figure 7). Before the return of a protected function is executed, the canary values are checked. If the canary values were not modified Stackguard implies that the programs integrity is assured (assuming that the canary values were altered, the program execution is aborted and an exception is issued). This approach protects only against stack smashing. However, there are still ways to overcome Stackguard. An attacker could guess the canary values. This is quite hard, if the values are chosen randomly for each guarded function, but it is also possible to choose a canary value made of terminator characters, which makes every string/file copy function to stop at the canary value. So even restoring the canary value would not lead to a successful program flow detour. Another way to thwart Stackguard is to find a pointer to overflow that it points to the address of the saved EIP and use that pointer as target for a copy function. This way the EIP is overwritten without modifying the canary values as it was shown by the authors in [22]. Overhead produced is moderate with up to 125% and that this method is not transparent, meaning that the source code is needed for recompilation. This fact makes Stackguard useless for many legacy software products on the market, because they are not open source.

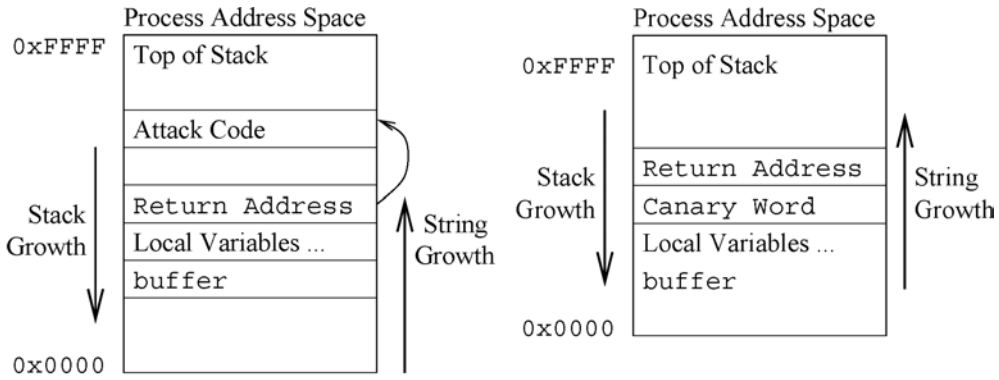

Figure 7: Shematic stack layout using Stackguard

3.1.2 Libsafe and libverify

Baratloo et al. propose in [1] two methods that should protect against buffer overflow attacks. The first method is libsafe, a transparent approach set up in a DLL that replaces standard (vulnerable) functions by standard bounds checked functions (e.g. strcpy could be replaced by strncpy). The upper limit of the bounds is calculated based on the EBP, so the maximum amount written to a buffer is the size of the stackframe[8] (see Figure 8). This method only works if the EBP can be determined, since there exist compiler options that make this impossible; further compatibility issues could arise with legacy software.

[7] There are two methods used in most cases, the one method chooses canary values randomly and the other method uses terminator values that stop string/file operations.

[8] Dependend on the start of the buffer on the stack.

Another problem is, that this approach only protects the EIP/EBP, so other attacks are still feasible.

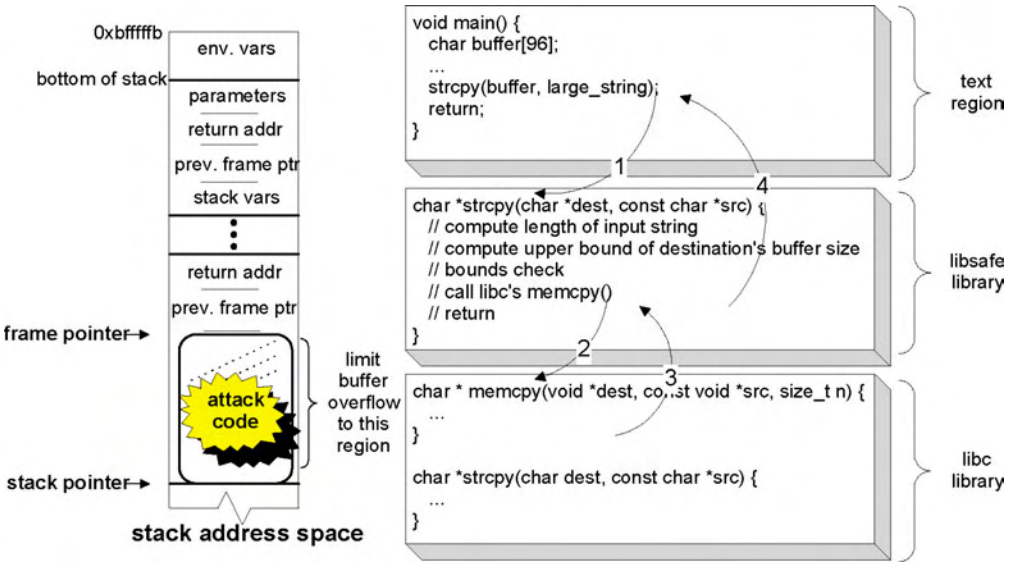

Figure 8: Libsafe function call and stack layout

The second method is similar to Stackguard and called libverify. It implements a wrapper function that saves the copy of the canaries to a canary stack. On a function call the function is copied to the heap, the first instruction gets overwritten by a jump to the wrapper entry function and the return instruction gets overwritten by the wrapper exit function (see Figure 9). This method is, such as Stackguard, vulnerable against attacks that do not smash the stack, same techniques can be applied to overcome the libverify protection.

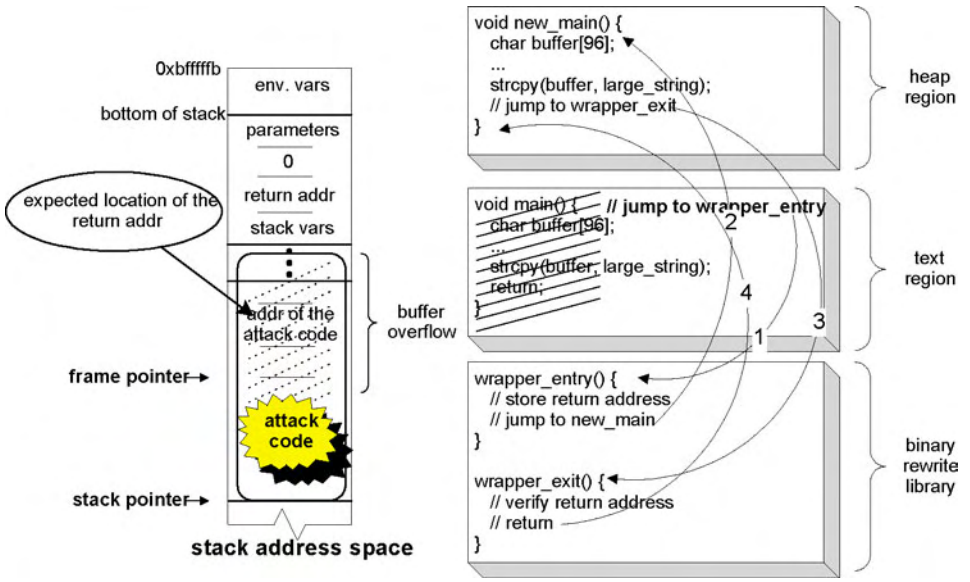

Figure 9: Libverify funtion call and stack layout

However, both protection methods are transparent, which make them suitable for most programs available. The overhead of Libsafe is very small ($\leq$ 15%) since only unsafe functions are intercepted. The overhead of Libverify is comparable to Stackguard protected programs ($\leq$ 125%). Both approaches are applicable in practice if not applied as the only protection method against buffer overflow attacks.

3.2 Instrumentation

3.2.1 Safe pointer

In [7] Austin et al. presented a so called *safe pointer* structure is to detect all pointer and array access errors. Meaning that both, temporal and spatial errors are detected. The structure consists of five entries:

- value (the value of the safe pointer, it may contain any expressible address)
- base (the base address of the referent)
- size (the size of the referent in bytes)
- storageClass (either Heap, Local or Global)
- capability (unique capability. Predefined capabilities are FOREVER and NEVER, else it could be an enumerated number as long as its value is unique)

Base and size are spatial attributes, capability and storageClass are temporal attributes. The capability is also stored in a capability store when it is issued and deleted if the storage is freed or when the procedure invocation returns. This ensures that storage that is not available (like freed heap allocated memory) is not accessed anymore. The transformation of a program from unsafe to safe pointers involves pointer conversion (to extend all pointer definitions), check insertion (to instrument the program to detect memory access errors) and operator conversion (to generate and maintain object attributes). Figure 10 shows the way safe pointers work with spatial and temporal access errors. This method protects the stack and the heap memory against overflowing of arrays. Attacks that are not prevented are off-by-one, since it is a typical programmers error and signed/unsigned overflows[9]. Another disadvantage is the overhead, since the size of the safe pointer itself is 15 bytes, the overhead is 275%. The text segment overhead ranges from 35% to 300% and the total size of the Data, BSS and heap segment has an overhead ranging from 5% up to 330%. With run-time optimizations[10] the overall execution overhead ranges from 130% to 540%. Last but not least this method is not transparent, because recompilation is needed to implement safe pointers.

[9] Typically, this attack is caused by misinterpreted integers, which is not caused by pointer or array accesses

[10] The authors give an example of a run-time optimization in their work.

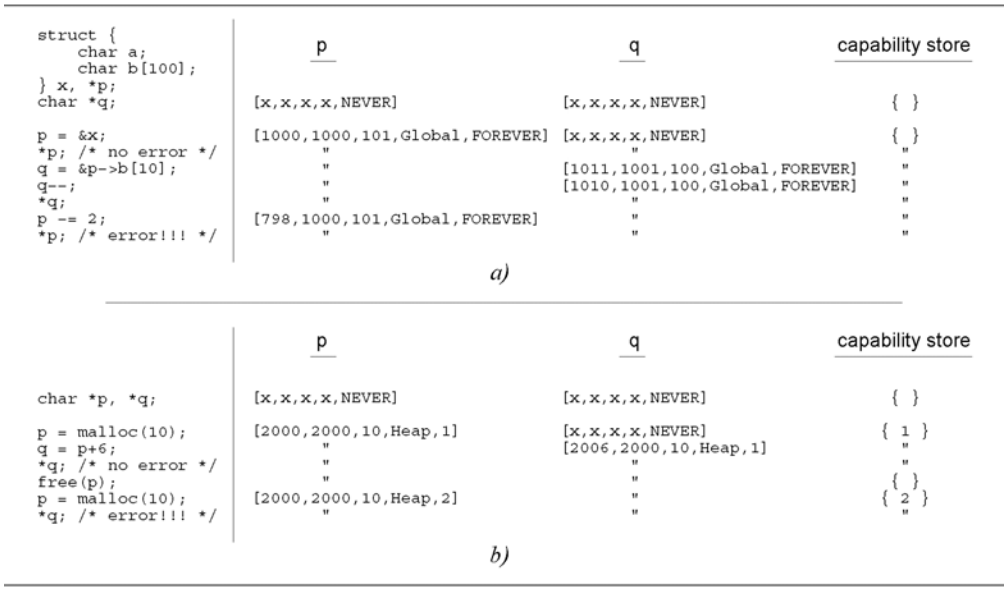

Figure 10: Memory access checking examples. Figure a) is an example of a spatial access error, Figure b) is an example of a temporal access error. Safe pointer values are specified as 5-tuple with the following format: [value,base,size,storageClass,capability]. An occurrence of x indicates a do not care value.

3.2.2 C Range Error Detector (CRED)

CRED is presented by Ruwase and Lam in [6] and presents the idea to replace every out-of-bounds (OOB) pointer value with the address of a special *OOB object* created for that value. To realize this, a data structure called *object table* collects the base address, and size information of static, heap and stack objects. To determine, if an address is in-bounds, the checker first locates the referent object by comparing the in-bounds pointer with the base and size information stored in the object table. Then, it checks if the new address falls within the extent of the referent object. If an object is out-of-bounds, an OOB object is created in the heap that contains the OOB address value and the referent object. If the OOB value is used as an address it is replaced by the actual OOB address. The OOB objects are entered into an *out-of-bounds object hash table* , so it is easy to check if a pointer points to an OOB object by consulting the hash table. The hash table is only consulted if the checker is not able to find the referent object in the object table or cannot identify the object as unchecked. Arithmetic and comparison operations to OOB objects are legal, since the referent object and its value is retrieved from the OOB object. But if a pointer is dereferenced, it is checked if the object is in the object table or if it is unchecked else the operation is illegal. Buffer overflows are not prevented but the goal is thwarted because copy functions need to dereference the OOB value, the program is halted before more damage happens. This fact could be still used as DoS attack, since the program (service) is halted and needs to be restarted or even worst if it has to be re-administrated. Figure 11 shows

schemetic how this method works, by comparing the non instrumented code, the CRED instrumented code and a code where the instrumentation is based on a previous approach by Jones and Kelly, which is the base for CRED. Since recompilation is needed, this method is not transparent. But the instrumented code is fully compatible to non-instrumented code[11]. The overhead of this approach ranges from 1% to 130%, but the authors do not show how certain kind of buffer overflow attacks, like signed/unsigned and off-by-one overflows (which were discussed earlier) are handled. The same arguments as on the safe pointers in the previous section can be applied here.

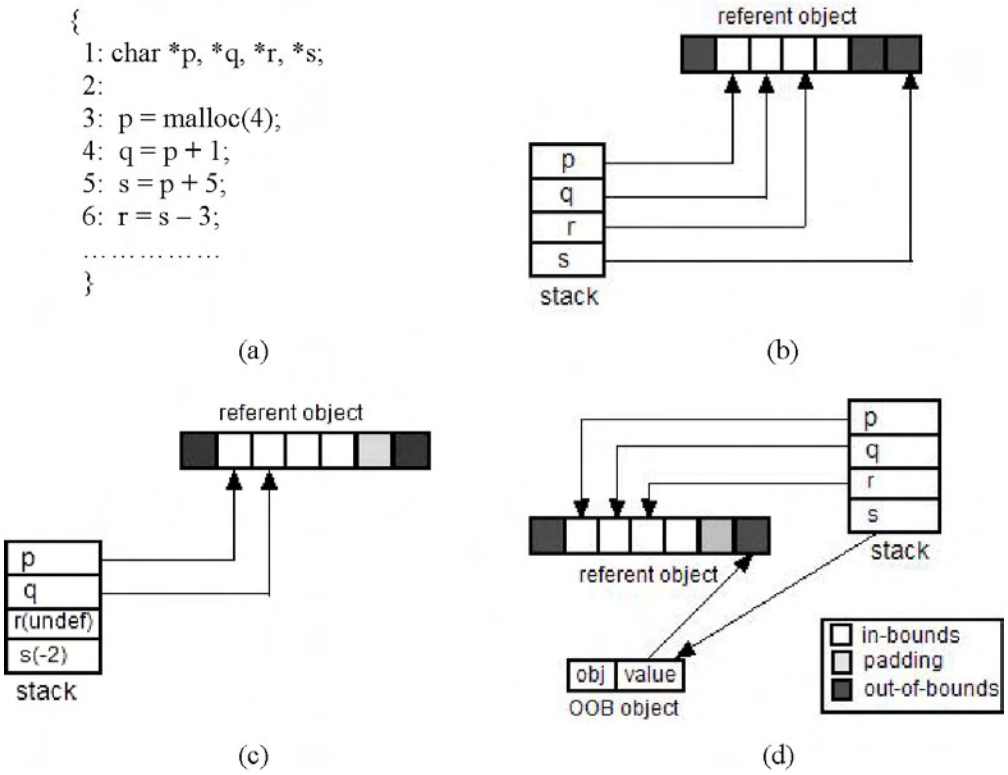

Figure 11: (a) Simple C program, memory states of (b) uninstrumented execution, (c) instrumentation with Jones and Kelly Checker, (d) instrumentation with CRED.

3.3 Hardware based

The approach by McGregor et al. in [4] deals with an architectural change. The authors suggest to implement a *Secure Return Address Stack* (SRAS), which is a cyclic, finite LIFO structure that stores return addresses. At a return call the last SRAS entry is compared with the return address from the stack and if the comparison yields that the return address was altered the processor can terminate the process and inform the operation system or continues the execution based on the SRAS return address. Since the SRAS is finite and

[11]This was the lack of Jones and Kellys approach, which broke working code.

cyclic, n/2 of the SRAS content has to be swapped on an under- or overflow. For this purpose the authors suggest two methods:

- OS-managed SRAS swapping. The operation system executes code that transfers contents to or from memory which is mapped to physical ages that can only be accessed by the kernel
- Processor-managed SRAS swapping. The processor maintains two pointers to two physical pages that contain spilled SRAS addresses and a counter that indicates the space left in the pages. If the Pages over- or underflow, the OS is invoked to de-/allocate pages, else the processor can directly transfer contents to and from the pages without invoking the OS

The performance impact of this approach is due to the swapping, since the hardware operation saving an address is small. The authors give examples for upper and lower bounds that could be used as stack size and with a "good" choice the performance impact of CPU swapping is about 1% or less and OS swapping has an impact of 1% up to 67,9%. Another shortcoming is that only the return address is protected, which prevents an attacker from redirecting the control flow. But heap overflows can be used to gain control of the program flow to, not to mention that other stack based attacks are still working. The authors also mention that their approach should be applied in conjunction with other countermeasures. The problem is that the SRAS is not compatible with non LIFO routines, such as C++ exception handling. This makes it necessary to change the non LIFO routines to LIFO routines or it must be possible to turn off the SRAS protection.

3.4 Static

The approach in [11] by Larochelle and Evans deals with the idea to comment the source code that LCLint can interpret them and generate a log file which can be used to identify possible vulnerabilities. If a source code is analyzed, LCLint evaluates conditions to fulfill safe execution of the finally compiled program. These conditions are written to the log file so the programmer can check if these conditions are true for every case that could happen while execution. Then the programmer can write control comments into the source to let LCLint what conditions are fulfilled or if LCLint should ignore parts of the source code. This way errors can be found before compilation, but this method has certain shortcoming. Since it is not possible to efficently determine invariants, the authors suggest to take advantage of idioms used typically by C programmers. Since this method is not exact, the rate of false positives grows. Further LCLint is a lightweight checker, meaning that the program flow is also checked using heuristics, since determining all possible program states might need exponential time[12]. The false positives can be commented out, but this means more work to

[12]This depends on the depth of the program, its size and the possible input, output and throughput and the amount of variables that could be modified and how they could be modified.

the developers of the software and since heuristics are used, false negatives are produced either. All these facts and the fact that this method is not transparent makes it only suitable for new or small projects. The last aspect we want to oint out is that this is the only method so far that produces no overhead, since the compilers skip comments.

3.5 Operation system based

3.5.1 Data Execution Prevention (DEP)

With the release of the Service Pack 2 for Windows XP and Service Pack 1 for Windows 2003 a new protection was introduced to machines using these operation systems, the DEP. In [20] Microsoft explains a bit how the DEP works. In cooperation with Intel (Execute Disable bit feature) and AMD (no-execute page-protection processor feature) a new CPU flag was implemented called the NX-Flag. It marks all memory locations in a process as non-executable unless the location explicitly contains executable code. If the machine running with DEP support has no NX-Flag, the DEP can be enforced by the operation system (software enforcement). This protection prevents execution of injected code, if the code was injected in a non executable area. As shown by Bulba and Kil3r in [23] the DEP can be bypassed. Further this method requires that even valid, working processes are (sometimes) recompiled[13]. Another shortcoming is, that this method does not prevent the buffer overflow itself, so attacks like variable attack or BSS/heap overflows are not prevented[14].

3.6 Solar Designer

The Solar Designer patch presented in [21] does nearly the same as the DEP, but it makes the stack non executable[15]. Since Linux needs the executable stack for signal handling, this restricts the normal behavior of Linux. If the attack is able to determine code that would act like a shellcode and execute this code instead of injected code[16] the patch can be bypassed. To conclude, buffer overflows are not prevented, only the code execution. Attacks like the variable attack or BSS/heap overflows are still possible, and heap overflows can also be used to execute arbitrary code.

3.7 Buffer overflow blocker

A method called SigFree, a signature free buffer overflow blocker is presented by Wang et al. in [5] a. It is a transparent approach working on the application

[13]Drivers or older applications sometimes try access executable code even if that code is located in a non executable part of the stack.

[14]Microsoft has also introduced a heap protection, but [23] shows that it is possible to overcome this protection as well.

[15]The patch also restricts links and FIFOs in /tmp, restricts /proc, destroys shared memory segments not in use and enforces RLIMIT_NPROC on execve(2).

[16]The attacker could analyze the attacked program or standard services to find code that would serve to fulfill his goal.

layer (Figure 13 shows the OSI Reference Model), that is layered between a service and the corresponding firewall. SigFree works as follows: First a request is decoded by the *URI decoder* to allow SigFree to find payloads located in the request-URI. After the decoding, the request is filtered by the *ASCII filter* , and if the query parameters of the request-URI and request-body are both printable ASCII ranging from 0x20-0x7E hexadecimal, SigFree allows the request to pass. Otherwise it is forwarded to the *instruction sequences distiller* . This module determines all possible instruction sequences from the query parameters of the request-URI and request-body. As last step the distilled request is handed over to the *instruction sequences analyzer* , which uses all the instruction sequences distilled by the instruction sequences distiller as input. This module analyzes the instruction sequences to determine whether one of them is (a fragment of) a program. If one of the instruction sequences is (a fragment of) a program, it is blocked, otherwise it is passed through (Figure 12 shows the SigFree architecture). Since valid requests contain code fragments[17], the amount of useful instructions found indicates if the request contains executable code. Therefore two schemes were developed: The first is based on the number of push calls found in a request and the second scheme is based on the detection of data flow anomalies called code abstraction. Under the assumption that services like web servers, remote access services and workstation services (and more) accept data only, blocking a request with more than 18 useful instructions would lead to block all attacks tested by the authors[18].

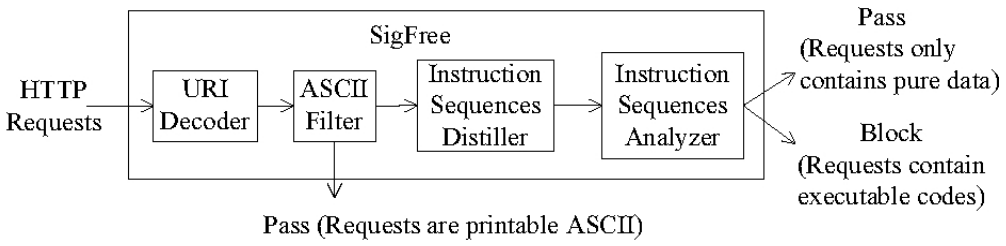

Figure 12: SigFree architecture

This approach is appropriate if the assumptions are fulfilled. The overhead is negligible, if the amount of attack messages is small. The usage of distillation of incoming request with following code abstraction enables SigFree to block even unknown attacks; the transparency of SigFree offers a wide, platform independent usage, so that existing applications can benefit from the protection without being modified. The flip side of the coin is that SigFree does not prevent buffer overflows. Every attack can be applied, which does not rely on code injection (e.g. variable attack, DoS); local buffer overflow attacks are not blocked at all. Even code execution can be done if likely code already exists[19]. Another problem is that shellcode might be small enough that it will not get

[17]After distillation some hexadecimal values can be interpreted as instruction sequences.

[18]The threshold number of push-calls should be set to 2, so scheme 1 can detect all attacks used in the tests. The threshold of useful instructions should be set to a number between 15 and 17, so scheme 2 can detect all the attacks used in the tests.

[19]See non-executable stack, same arguments.

blocked[20]. Further SigFree cannot be applied to encrypted communications. The authors suggest to implement SigFree as plug-in for the services that use such encrypted communication (e.g. Apache would need SigFree as module if it should be applied to SSL secured communication). Last but not least the assumptions (that incoming requests contain no executable content) must be fulfilled. On a client, valid incoming packets would be blocked if they contain executable code. This would happen on every meta file (picture, video or sound), stream (VoIP, conferencing etc.) and on every download that contains executable data. This could also become a problem on a protected server, because if a protected service can be used for uploads, some uploads would be blocked if they contain (legal) executable contents (e.g. the upload of pictures, sounds, videos or even applications that should be offered via the internet might be blocked if they contain meta informations).

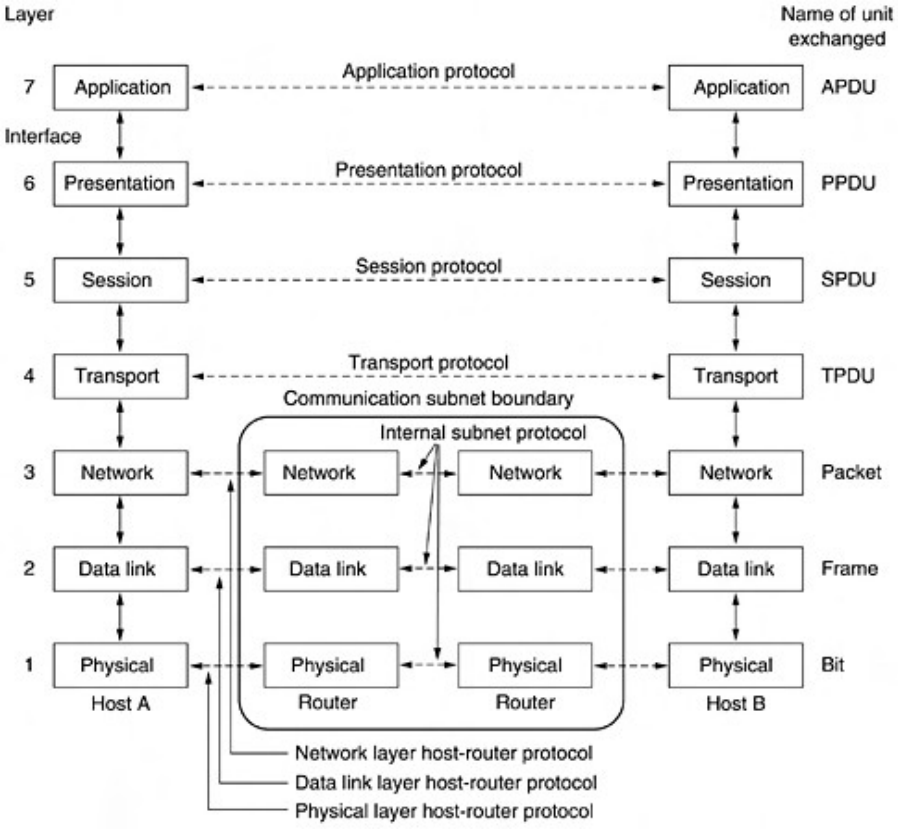

Figure 13: The OSI Reference Model

[20]See code obfuscation in the basics, the example of the encryption/decryption can be modified, that the shellcode is small enough to be not blocked (prerequisite is the threshold used to configure SigFree).

4 Conclusions

As we pointed out, it takes considerable effort to prevent buffer overflows. On the one hand static methods produce false positives/negatives, which causes manual corrections in the source code by the developers. On the other hand instrumentation methods have a lot of overhead and they are not transparent. With a little more research and some optimizations of the architecture (hardware and software) they prevent most of the buffer overflow attacks. Stack based methods do not protect against all attacks and we determined that some protections can be bypassed. Hardware methods provide protection with little overhead, but architectural changes have to go deeper than we've seen with the SRAS approach. Making the stack non-executable does not solve the buffer overflow problem either, it is just more difficult to launch a successful attack. Finally blocking buffer overflows based on code injections does not prevent most of the attacks and the assumptions must be met, else valid communications could be blocked.
The protection against all introduced attacks is too dynamic for the current architecture, thus it is expensive to protect current systems against known buffer overflow attacks. There are solutions for this problem, e.g. the architecture could be changed to make buffer overflows impossible or very hard to misuse (the SRAS approach shows one example). The operation systems should be taken in account too, because serious changes have been made in the last years, however, they were still workarounds. Like the safe pointer approach showed, with some overhead buffer overflow attacks could be prevented. This fact is not very surprisingly, because all buffer overflows are caused by errors made by programmers and/or data processing. This leads to the conclusion that data processing has to be renewed consequently. For this purpose the C standard should be updated. The vulnerable functions that are used for buffer overflows are well known today and the countermeasures show that many ideas have been made to patch this issue. Updating the C standard would benefit from all researches made so far[21] and it could be implemented with downwards compatibility, so existing applications could be recompiled and benefit from the update without being changed or rewritten.

References

[1] Baratloo, A., Singh, N., and Tsai, T. Transparent run-time defense against stack smashing attacks. In Proceedings of the 9th USENIX Security Symposium, June 2000.

[2] Collberg, C., Thomborson, C., and Low, D. A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science,University of Auckland, July 1997.

[3] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. Stackguard: Automatic adaptive

[21] Like optimizations and safe functions.

detection and prevention of buffer-overflow attacks. In Proceedings of 7th USENIX Security Conference, January 1998.

[4] McGregor, J., Karig, D., Shi, Z., and Lee, R. A processor architecture defense against buffer overflow attacks. In Proceedings of International Conference on Information Technology: Research and Education (ITRE) (2003).

[5] Wang, X., Pan, C., Liu, P., and Zhu. S. SigFree: A Signature-free Buffer Overflow Attack Blocker. To appear in USENIX Security'06, July 2006.

[6] Ruwase, O., and Lam, M. S. A Practical Dynamic Buffer Overflow Detector. In Proceedings of the Network and Distributed System Security (NDSS) Symposium, February 2004.

[7] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1994.

[8] Neuhaus, S. Buffer Overflows und Lösungen dazu. May 2003.

[9] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In proceedings of the 10th USENIX Security Symposium, August 2001.

[10] Shankar U., Talwar K., Foster, J. S., and Wagner D. Detecting Format String Vulnerabilities with Type Qualifiers. In Proceedings of the 10th USENIX Security Symposium, August 2001.

[11] Larochelle, D., and Evans, D. Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, August 2001.

[12] Alaph1. Smashing stack for fun and profit. Phrack Magazine, Volume 7, Issue 49, November 1996.

[13] Opatz, F. Buffer Overflows für Jedermann, July/August 2005.

[14] Klein, T. Buffer Overflows und Format-String-Schwachstellen. Dpunkt Heidelberg, 2004, ISBN 3-89864-192-9.

[15] Hyde, R. The Art of Assembly Language Programming. http://webster.cs.ucr.edu/AoA/index.html

[16] Shao, Z., Xue, C., Zhuge, Q., Sha, E.H.-M., and Xiao B. Security Protection and Checking for Embedded System Integration Against Buffer Overflow Attacks via Hardware/Software. In Proceedings of Information Assurance and Security special track in conjunction with the International Conference on Information Technology: Coding and Computing (ITCC 2004).

[17] Zachmann, G. Programmierung I, Exkurs: Buffer Overflows, University Bonn

[18] Friedrich, M. Buffer Overflows, Heap Overflows, Format String-Fehler und Race Conditions, Februar 2006

[19] Leidecker, N. Ausführungssperren von Programmcode, Januar 2006

[20] Andersen S., and Abella, V. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, September 2004

[21] Solar Designer. Linux kernel patch from the Openwall Project, http://www.openwall.com/linux/README.shtml

[22] Bulba and Kil3r. Bypassing Stackguard and Stackshield. Phrack Magazine Volume 10, Issue 56, May 2000.

[23] Anisimov, A. Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass, 2004.

[24] Tanenbaum, A. S. Computer Networks, Fourth Edition. Prentice Hall, March 2003, ISBN 0-13-066102-3.